Nola Local - Database Model

Database Models Overview

Collections:

1. **users** - User accounts and authentication
2. **events** - All events (user-created and external)
3. **categories** - Event categories (small, rarely changes - could be embedded but separate for reusability)

---

1. User Model

```
{
  _id: ObjectId("U1111"),

  // Authentication fields
  username: "string" (unique, required, indexed),
  email: "string" (unique, required, indexed),
  password: "string" (hashed, required),

  // Email verification
  isVerified: Boolean (default: false),
  verifyToken: "string" (nullable),
  verifyTokenExpiry: Date (nullable),

  // Password reset (stretch goal)
  resetToken: "string" (nullable),
  resetTokenExpiry: Date (nullable),

  // References to events (array won't grow unbounded - users typically create/like limited events)
  createdEvents: [
    ObjectId("E1111"),
    ObjectId("E1112")
  ],

  likedEvents: [
    ObjectId("E2221"),
```

```
    ObjectId("E2222")
  ],

  // Timestamps
  createdAt: Date,
  updatedAt: Date
}
```

Why this structure:

- **Don't embed events** - Events need to be accessed independently, shared across users, and filtered globally
- **Array of ObjectIds** - Won't grow unbounded (reasonable limit: users typically create < 100 events, like < 500 events)
- **Indexed fields** - username, email for fast authentication lookups
- **Separate verification fields** - Easy to query unverified users, easy to clear after verification

Potential issue & mitigation:

- ⚠️ If `likedEvents` array could grow very large (>1000), consider separate `likes` collection
- For now, keeping it embedded is fine for MVP

---

2. Event Model (Core - Most Complex)

```
{
  _id: ObjectId("E1111"),

  // Basic event info
  title: "string" (required, indexed for text search),
  description: "string" (required, indexed for text search),

  // Date/Time
  date: Date (required, indexed),
  time: "string" (e.g., "7:00 PM"),

  // Location
  location: "string" (required),

  // Category - REFERENCE not embed (compelling reason: categories accessed independently for filters)
```

```
  category: ObjectId("C1111") (references Category),

  // Image
  imageUrl: "string" (Cloudinary URL or external API URL),

  // Source tracking
  source: "string" (enum: 'user', 'eventbrite', 'ticketmaster', indexed),
  sourceUrl: "string" (nullable, link to original event),
  externalId: "string" (nullable, for deduplication, unique compound index with source),

  // Creator - REFERENCE not embed (compelling reason: user data changes, need to access
user independently)
  creator: ObjectId("U1111") (nullable - null for external events),

  // Likes - EMBEDDED array of ObjectIds (won't grow unbounded - realistic max ~1000 likes
per event)
  likes: [
    ObjectId("U1111"),
    ObjectId("U1112"),
    ObjectId("U1113")
  ],

  // Denormalized count (avoid counting array every time)
  likesCount: 0 (integer, updated when likes array changes),

  // Status
  status: "string" (enum: 'upcoming', 'passed', indexed),

  // Metadata
  createdAt: Date,
  updatedAt: Date,
  lastSyncedAt: Date (nullable, for external events only)
}
```

Why this structure:

- ✅ **Reference category** - Categories are accessed independently for filtering, dropdowns
- ✅ **Reference creator** - Users need to be queried independently, user data might change
- ✅ **Embed likes array** - Bounded growth (reasonable event won't have >10,000 likes), fast access
- ✅ **Denormalize likesCount** - Avoid array.length calculation on every query (trade-off: slight complexity updating count)

- ✅ **Text indexes on title/description** - Enable search functionality
- ✅ **Compound index on (externalId, source)** - Prevent duplicate external events

Indexes:

```
// Compound index for deduplication
{ externalId: 1, source: 1 }, { unique: true, sparse: true }

// Query optimization indexes
{ date: 1, status: 1 }
{ category: 1, status: 1 }
{ source: 1 }
{ creator: 1 }

// Text search
{ title: "text", description: "text" }
```

Potential issue & mitigation:

- ⚠️ If an event goes viral and gets >10,000 likes, `likes` array approaches 16MB limit
- 💡 **Solution**: Move to separate `likes` collection if this becomes real (unlikely for local events app)
- For MVP, embedded array is much more performant

---

3. Category Model

```
{
  _id: ObjectId("C1111"),

  name: "string" (required, e.g., "Live Music"),
  slug: "string" (required, unique, indexed, e.g., "live-music"),
  color: "string" (hex color for UI, e.g., "#FF6B6B"),

  // Pre-seeded, rarely changes
  createdAt: Date
}
```

Why separate collection (not embedded in events):

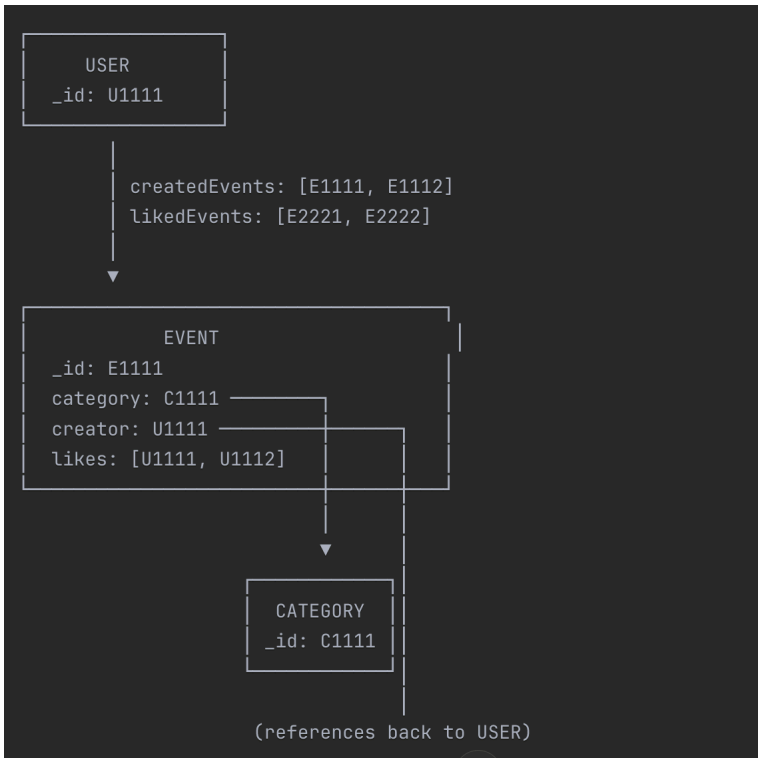
- ✅ **Compelling reason**: Categories accessed independently for filter dropdowns, navigation

- ✅ **Small collection** (~6-10 categories total), frequently read, rarely written
- ✅ **Consistency** - Easy to update category name/color once, reflects everywhere
- ⚠️ **Could embed** - But referencing is better for data consistency and filter queries

Pre-seeded categories:

```
[
  { name: "Live Music", slug: "live-music", color: "#FF6B6B" },
  { name: "Food & Drink", slug: "food-drink", color: "#4ECDC4" },
  { name: "Arts & Culture", slug: "arts-culture", color: "#FFE66D" },
  { name: "Community", slug: "community", color: "#95E1D3" },
  { name: "Sports", slug: "sports", color: "#F38181" },
  { name: "Other", slug: "other", color: "#AA96DA" }
]
```

Relationships Diagram



Deduplication Strategy for External Events

Problem: Same event might appear on Eventbrite AND Ticketmaster

Solution: Compound unique index on `(externalId, source)`

```
// This prevents duplicates within same source
{ externalId: "evt_123", source: "eventbrite" } ✅
{ externalId: "evt_456", source: "eventbrite" } ✅
{ externalId: "evt_123", source: "eventbrite" } ❌ Duplicate!

// But allows same event ID from different sources
{ externalId: "evt_789", source: "eventbrite" } ✅
{ externalId: "evt_789", source: "ticketmaster" } ✅ Different source, allowed
```

Additional fuzzy deduplication (in sync controller):

```
// Before inserting external event, check for similar events:
const potentialDuplicate = await Event.findOne({
  title: { $regex: new RegExp(escapeRegex(title), 'i') },
  date: { $gte: startOfDay(date), $lte: endOfDay(date) },
  location: { $regex: new RegExp(escapeRegex(location), 'i') }
});

if (potentialDuplicate) {
  // Skip or merge
}
```

Data Access Patterns & Query Examples

1. Get all upcoming events with filters

```
Event.find({
  status: 'upcoming',
  category: categoryId,
  date: { $gte: startDate, $lte: endDate }
})
.populate('category', 'name slug color')
.populate('creator', 'username')
.sort({ date: 1 })
.limit(50);
```

Why efficient: Indexed on `status`, `category`, `date`

---

2. Get event with like status for current user

```
const event = await Event.findById(eventId)
  .populate('category', 'name slug color')
  .populate('creator', 'username');

const isLikedByUser = event.likes.includes(currentUserId);
const isCreator = event.creator?._id.toString() === currentUserId;
```

Why efficient: No JOIN needed, likes embedded in document

---

3. Get user's created events

```
Event.find({ creator: userId, status: 'upcoming' })
  .populate('category', 'name slug color')
  .sort({ date: 1 });
```

Why efficient: Indexed on `creator`

Alternative using embedded array (less efficient, avoid):

```
const user = await User.findById(userId).populate('createdEvents');
```

❌ This is worse - requires loading user first, then populating potentially many events

---

4. Get user's liked events

```
Event.find({
  _id: { $in: user.likedEvents },
  status: 'upcoming'
})
.populate('category', 'name slug color')
.sort({ date: 1 });
```

Why acceptable: User's likedEvents array is bounded, this is essentially an indexed lookup

---

5. Toggle like on event

```
// Unlike
await Event.findByIdAndUpdate(eventId, {
  $pull: { likes: userId },
  $inc: { likesCount: -1 }
});

await User.findByIdAndUpdate(userId, {
  $pull: { likedEvents: eventId }
});

// Like
await Event.findByIdAndUpdate(eventId, {
  $addToSet: { likes: userId },
  $inc: { likesCount: 1 }
});

await User.findByIdAndUpdate(userId, {
  $addToSet: { likedEvents: eventId }
});
```

Trade-off: Requires 2 DB operations, but keeps data consistent

---

Handling Array Growth Concerns

When to worry about array size:

Safe (embedded arrays):

- ✅ `user.createdEvents` - Users rarely create >100 events (100 ObjectIds = ~1.2KB)
- ✅ `user.likedEvents` - Even power users unlikely to like >1000 events (1000 ObjectIds = ~12KB)
- ✅ `event.likes` - Local events rarely get >1000 likes (1000 ObjectIds = ~12KB)

Document size math:

- 1 ObjectId = 12 bytes
- 1000 ObjectIds = ~12KB
- 10,000 ObjectIds = ~120KB
- 100,000 ObjectIds = ~1.2MB
- MongoDB limit = 16MB

If arrays grow too large (future optimization):

Create separate `Likes` collection:

```
{
  _id: ObjectId,
  userId: ObjectId (indexed),
  eventId: ObjectId (indexed),
  createdAt: Date
}

// Compound unique index
{ userId: 1, eventId: 1 }, { unique: true }
```

Query becomes:

```
const likedEventIds = await Like.find({ userId }).distinct('eventId');
const events = await Event.find({ _id: { $in: likedEventIds } });
```

But for MVP, embedded arrays are **much more performant**.

Schema Validation (Mongoose)

User Schema

```
const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    minlength: 3,
    maxlength: 30
  },
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true,
    validate: [validator.isEmail, 'Invalid email']
  },
```

```
  password: {
    type: String,
    required: true,
    minlength: 8
  },
  isVerified: {
    type: Boolean,
    default: false
  },
  verifyToken: String,
  verifyTokenExpiry: Date,
  resetToken: String,
  resetTokenExpiry: Date,
  createdEvents: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Event'
  }],
  likedEvents: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Event'
  }]
}, {
  timestamps: true
});

// Indexes
userSchema.index({ email: 1 });
userSchema.index({ username: 1 });
```

Event Schema

```
const eventSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
    trim: true,
    maxlength: 200
  },
  description: {
    type: String,
    required: true,
    maxlength: 2000
  },
  date: {
```

```
    type: Date,
    required: true
  },
  time: String,
  location: {
    type: String,
    required: true
  },
  category: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Category',
    required: true
  },
  imageUrl: String,
  source: {
    type: String,
    enum: ['user', 'eventbrite', 'ticketmaster'],
    required: true
  },
  sourceUrl: String,
  externalId: String,
  creator: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  },
  likes: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }],
  likesCount: {
    type: Number,
    default: 0
  },
  status: {
    type: String,
    enum: ['upcoming', 'passed'],
    default: 'upcoming'
  },
  lastSyncedAt: Date
}, {
  timestamps: true
});

// Indexes
```

```
eventSchema.index({ externalId: 1, source: 1 }, { unique: true, sparse: true });
eventSchema.index({ date: 1, status: 1 });
eventSchema.index({ category: 1, status: 1 });
eventSchema.index({ source: 1 });
eventSchema.index({ creator: 1 });
eventSchema.index({ title: 'text', description: 'text' });
```

Category Schema

```
const categorySchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    unique: true
  },
  slug: {
    type: String,
    required: true,
    unique: true,
    lowercase: true
  },
  color: {
    type: String,
    required: true,
    match: /^#[0-9A-F]{6}$/i
  }
}, {
  timestamps: true
});

categorySchema.index({ slug: 1 });
```

Summary: Why This Design

✅ Follows MongoDB best practices:

1. Embed `likes` in events - bounded array, fast access
2. Reference `category` and `creator` - need independent access
3. Arrays won't grow unbounded - realistic limits on user-generated content
4. Denormalize `likesCount` - performance optimization
5. Designed for app's unique needs - event discovery with social features

✅ **Prevents duplicates:**

- Unique indexes on username, email
- Compound unique index on (externalId, source)
- Fuzzy matching logic for cross-platform duplicates

✅ **Optimized for common queries:**

- All frequent queries use indexes
- Minimal population needed
- Embedded data reduces JOINs

✅ **Scalable:**

- Clear migration path if arrays grow too large
- Indexes support all filter/search operations
- Can shard on date if event volume grows massively